# **CSEP 517, Fall 2015, Assignment 3**

<https://github.com/jogonzal-msft/UWNLPAssignment3>

## Introduction

The task is as follows – given an existing inefficient BaseLineParser, improve it to be a real CKYParser.

The BaseLineParser does a pretty crazy process and does not obtain really good results – it takes a sentence, tags its words with the most likely tag, and then looks for that tag in the training set. It iterates by right branching, so it always builds trees that are heavily biased to one side. The scores for this tagger are not great – see results.

The CKYParser should use the CKY (Cocke–Younger–Kasami) algorithm to find the most efficient tree for a given sentence according to training data.

## Implementing the CKY parser

The entire implementation of the CKYParser can be found in PCFGParserTester.java, in the CKYParser class. A few other classes were developed to aid with this algorithm (mainly container classes):

|  |  |
| --- | --- |
| Class | Purpose |
| Cell | Represent one cell in the CKY tree |
| BinaryCell | A cell with Binary options in the CKYTree |
| PreterminalCell | A cell with Preterminal options in the CKYTree |
| BinaryTag | Represents an individual binary tree structure with a score (one row in a BinaryCell) |
| PreterminalTag | Represents an individual preterminal tree structure with a score (one row in a PreterminalCell) |
| TagWithPosition | Container class |
| Position | Container class |

The CKYParser gets initialized in the constructor and then is called once per sentence on getBestParse. Here are the algorithm steps for that method:

1. For each word, compute all possible preterminal tags
   1. Compute unary closures for those tags
2. Store PreterminalCells in the first diagonal of the matrix (bottomUpTree)
3. Iterate to the next diagonal of BinaryCells for binary combinations
   1. Find all combinations according to grammar and fill out BinaryCell
   2. Find unary closures for that cell
4. When reaching the corder of the matrix
   1. Find S and traverse the tree down

Traversing the tree down isn’t as easy. Here are the steps:

1. Given the first binary of the tree
2. Add that tag to the tree, and create a new list of trees under it
3. Traverse down all of the unaries of this binary, until reaching the binary
4. When reaching the binary, recursively repeat this process (from (2))) for both nodes
5. When reaching a preterminalCell
   1. Traverse down all of the unaries, until reaching the preterminal
   2. Add the preterminal and the word

## Handling unary transitions

An algorithm similar to the one proposed in [Dan Jurafky’s video explaining CKY](https://www.youtube.com/watch?v=MiEKnFyErbQ') was used. For every cell, we store a backpointer Map<String, String> which knows which tag to redirect to in unary transitions. “Fake” rows or “PreterminalTag” and “BinaryTag” (with IsUnary=true) are created to signal that the unaryTransition needs to happen when the tree will be built.

Existing transitions might be replaced by other unary transitions (when they are better), making multiple backpointers possible.

## Results + Starting with “S”

Certain sentences do not begin with “S” in the corpus. Therefore, I decided to make a test with the following variants:

1. BaseLineParser
2. CKYParser always choosing “S” at start
3. CKYParser with STip
   1. STip is a Boolean passed in to “GetBestParse” which tells GetBestParse whether the sentence begins with “S” or not. If it begins with S. This information means that the parser will get the first tag (S) correctly if it’s an S, and for others it will simply pick the most likely tag.
4. (bonus) CKYParser with STip + horizontal markov
5. (bonus) CKYParser with STip + vertical markov
6. (bonus) CKYParser with STip + horizontal + vertical markov

Here are the results (including performance of algorithm) that were obtained running the parser with the following configurations: **maxTrainLength=1000, maxTestLength=40**.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | Performance per sentence (ms) | P | R | F1 | EX |
| (A) BaseLineParser | 0.92 | 15.43 | 17.49 | 16.4 | 2.31 |
| (B) CKYParser always choosing “S” at start | 7328.44 | 67.61 | 56.22 | 61.39 | 2 |
| (C) CKYParser with STip | 7496.26 | 75.62 | 66.39 | 70.71 | 6.59 |
| (D) CKYParser with STip + horizontal markov | 15823.42 | 76.87 | 70.58 | 73.59 | 8.6 |
| (E) CKYParser with STip + vertical markov | 16004.76 | 81.22 | 73.67 | 77 | 10.06 |
| (F) CKYParser with STip + horizontal + vertical markov | 18345.39 | 81.03 | 75.74 | 78.3 | 10.65 |

(Looking at F1) As expected, (B) outperforms (A). (C) outperforms (B) since it has more information about the first tag in the tree. (D) and (E) outperform (C) since they use grammar refinement (Markov). (F) outperforms (D) and (E) since it is a combination of both vertical and horizontal Markov.

## Common errors

Common errors that (C) and (D) make are choosing the wrong root – this is also be because of the reasons cited below. Looking more closely at (B) (which most of the time chooses the right root), common errors frequently fall into the description below:

Syntactic ambiguities and dark ambiguities make it hard for the tree parser to get the right result. There are often many ways of correctly building a tree, and sometimes one of them occurs more often, even though it is not the way humans meant that sentence. One example is the sentence below (see the occurrence of the word “itself”) the CKY algorithm isn’t sure whether to put “itself” below or above “than”. In Human language, the words “itself” and “than” frequently go together, but because no vertical/horizontal markovization has been applied, the tree has no notion of this and ends up applying “itself” to the wrong part of the sentence, thus yielding an F1 of ~53.

Sentence:

Stock-index futures contracts settled at much lower prices than indexes of the stock market itself .

Guess:

(ROOT

(S

(NP (JJ Stock-index) (NNS futures) (NNS contracts))

(VP (VBN settled)

(PP (IN at)

(NP

(NP

(ADJP (RB much) (JJR lower))

(NNS prices))

(PP (IN than)

(NP

(NP (NNS indexes))

(PP (IN of)

(NP (DT the) (NN stock) (NN market)))))))

(PRP itself))

(. .)))

Gold:

(ROOT

(S

(NP (JJ Stock-index) (NNS futures) (NNS contracts))

(VP (VBD settled)

(PP (IN at)

(NP

(NP

(ADJP (RB much) (JJR lower))

(NNS prices))

(PP (IN than)

(NP

(NP (NNS indexes))

(PP (IN of)

(NP

(NP (DT the) (NN stock) (NN market))

(NP (PRP itself)))))))))

(. .)))

Sentence processing MS: 2220. Average 1604.7976383225807

[Current] P: 58.33 R: 50.0 F1: 53.84 EX: 0.0

[Average] P: 79.25 R: 71.33 F1: 75.08 EX: 12.9

Other common found errors are similar ambiguities – and should be improved with vertical/horizontal markovization.

## Extra points – Horizontal/Vertical markovization

To achieve markovization, the class TreeAnnotationsMarkov was created. It contains the VerticalMarkov and HorizontalMarkov recursive methods, which simply walk the graph, appending the “=” and “-“ symbols next to parent tag and sibling tag respectively. The unAnnotateTree method takes care of removing those added symbols.

## Testing/development notes

The reported performance numbers were obtained when running with the following configuration on an 8 core 32 gig machine against the test set:

-server -mx1152m -Xmx1152m -XX:+UseConcMarkSweepGC -XX:+UseParNewGC -XX:+CMSConcurrentMTEnabled

To run the program, simply comment in/out commands from the main class (note that the program did take around ~7 hours to finish)

**public static void** main(String[] args) {  
 System.***out***.println(**"Hello World!"**);  
 Test<MockClass> test = **new** Test<MockClass>();  
 test.*main*(**new** String[]{  
 **"-test"**,  
 **"-maxTrainLength"**, **"1000"**,  
 **"-maxTestLength"**, **"40"**,  
 **"-usestip"**,  
 **"-quiet"**,  
 **"-horizontalmarkov"**,  
 **"-verticalmarkov"** });  
}